PARALLEL METHODS FOR THE GENERATION OF PARTITIONED INVERTED FILES

[1]A. MacFarlane, [2]J.A.McCann, [1,3] S.E.Robertson
[1] Centre for Interactive Systems Research, City University, London
[2]Department of Computing, Imperial College London
[3]Microsoft Research Ltd, Cambridge

Abstract: The generation of inverted indexes is one of the most computationally intensive activities for information retrieval (IR) systems: indexing large multi-gigabyte text databases can take many hours or even days to complete. We examine the generation of partitioned inverted files in order to speed up the process of indexing. We describe the components of PLIERS, the system used to index the documents and how these components can be re-configured to generate indexes with different types of partitioning. Two types of index partitions are investigated: *TermId* and *DocId*. Two types of build are investigated: *local* and *distributed*. The results from runs on both partitioning and build methods are compared and contrasted, concluding that *DocId* is the more efficient method.

1. INTRODUCTION

The generation of inverted indexes for text databases is a computationally intensive process that requires the exclusive use of processing resources for long periods. The following considers techniques that could be used in order to speed up the generation of the initial inverted file. The research described in this paper is part of an overall effort to understand and quantify the effects that differing partitioning methods for inverted files in parallel IR systems have on the performance of indexing, search, passage retrieval and index update (MacFarlane, 2000). Two types of partitioning methods are investigated: term identifier (*TermId*) partitioning and document identifier (*DocId*) partitioning: a partition is defined as the logical distribution of the inverted file. *TermId* partitioning is a type of partitioning which distributes each word to a single partition, while *DocId* partitioning distributes each document to a single partition. These partitions are fragmented across physical disks. A fuller discussion of these partitioning methods can be found in (Jeong & Omiecinski, 1995; MacFarlane et al, 1997) and an example can be found in appendix 1. Two types of index build methods are used: *Local* and *Distributed*. With *local build*, documents are kept on a *Local* disk and analysis is done on that *Local* disk only (Hawking, 1997): this method is applicable to *DocId* partitioning only. The *distributed build* method works by distributing the documents to nodes from a single disk. Section 2 describes a re-configurable process topology used to create different types of partitioned inverted files. Sections 3 and 4 describe the individual components of this process topology, while the indexing methodology used for the experiments is outlined in section 5. The hardware used for the experiments is described in section 6 while the data used in the experiments is described in section 7. In sections 8 and 9 we describe some results on build methods using *DocId* partitioning and *TermId* partitioning respectively, concluding in section 10 by comparing and contrasting the results. We provide a glossary of terms at the end.

2. INDEXING TOPOLOGIES

Our requirement for indexing topologies is to be able to support both partitioning methods under consideration as well as the two build strategies. The components of the topology must be reconfigurable in order to create different build types and numbers of inverted file partitions using different process combinations. Figure 1 show examples of both types of builds using the *DocId* partitioning method, together with process to processor mapping examples.

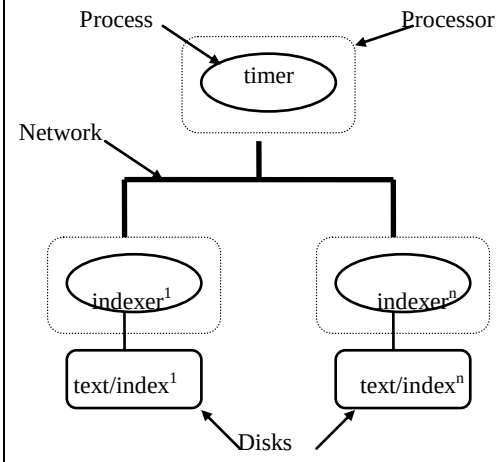


Fig 1a - *local build*

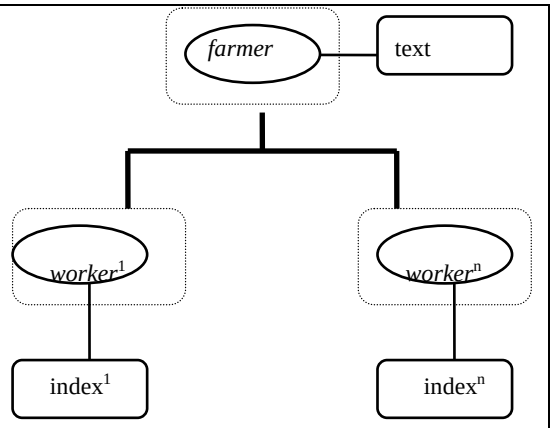


Fig 1b - *distributed build*

Fig 1 - Build examples using the *DocId* partitioning method

The *local build* method for parallel indexing is a very simple topology requiring little communication (see fig 1a). Each indexer node runs independently with no need for communication between them (the function of the indexer is described below). This form of build is applicable to *DocId* only. The *distributed build* method uses the process farm paradigm (Bowler et al, 1989) and an example of the one proposed for indexing is shown in fig 1b. The structure in the example consists of a *farmer* and n *worker* processes whose function is described below. Fig 1 shows the contrast in the build methods particularly with regard to the distribution of text to be indexed. The difference between the two methods is that text is kept locally when the *local build* method is used, and kept centrally on a single disk when *distributed build* is used (see appendix 2 for an example of how this works). We use *local build* where a given collection could not be physically placed on a single disk (e.g. VLC2/WT100g).

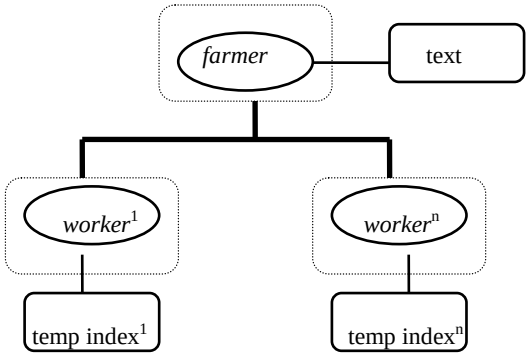


Fig 2a - Distribution of text Phase

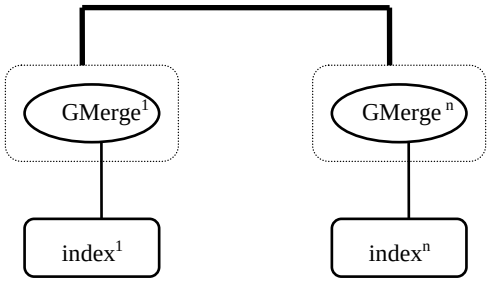


Fig 2b - Distribution of data to paritition phase

Fig 2 - Distributed Build example using the *TermId* partitioning method

Each method has its own advantages and disadvantages, and we leave the detailed discussion of such until later. The issue of communication is important here. It can be seen from both the diagrams and the descriptions above that some topologies will require a great deal more network resource than others. For example *distributed build* methods will require more communication than *local build* indexing in order to distribute text. Fig 2 shows an index topology example for *TermId* partitioning.

3. *DISTRIBUTED BUILD* TOPOLOGY COMPONENTS

In this section we describe the functions of the *farmer*, *worker* and *global merge* parallel processes. Note that there is only one *farmer* processor, and a number of *worker* processes (which become *global merge* processes in *TermId*). Our reason for using this method is that it allows us to automatically distribute text to nodes: it has the disadvantage in that the method is more communication intensive than the *local build* method (see section 4).

```
Distributed Initial set of documents/files to all workers

Loop no of files/documents
        get a request from worker i
        Case(request type)
            work request: send document/file to worker i
             id request     : send block of document id's to worker i
        EndCase
EndLoop
Loop until all workers have been terminated
        get a request from any worker i
        Case(request type)
            work request: send termination notice to worker i
             id request     : send block of document id's to worker i
        EndCase
EndLoop
```

Fig 3. Farmer algorithm for parallel indexing

3.1 Farmer Process

The *farmer*'s job is to distribute documents to the *worker*s (see Fig 3). Essentially it distributes work as equally as possible to create the least amount of load imbalance possible. Single documents or files containing multiple documents can be distributed: the latter saves communication time. There is an initialisation stage where each *worker* is given its first initial document/file; after that *worker*s are only given documents/files when they request them, i.e. send a message to the *farmer* asking for more work. When no more documents/files are left, a termination notice is sent to every *worker* process. Document identifiers are allocated individually if the granularity of parallelism is documents and in blocks if it is files. The document length cannot be recorded until the document has been analysed, and this data is sent to the *farmer* when a *worker* requests further work: this data is saved to disk when received. In an attempt to keep *worker*s load balanced a request for work is serviced as soon as possible after it has been received so that *worker*s who index small documents or files are not kept waiting for too long.

3.2 Worker Process

The *worker*'s function is to break down the document into its constituent parts, i.e. terms, and perform some analysis on these terms, e.g. stemming using the methodology described below (see fig 4). If required, the position record is stored for each term using current values of accumulated data for field number, paragraph number, sentence number, word number and preceding stop words. After each word is found these values are updated. The *worker* creates and inserts this word/position data in a bucket: the method of storage for bucket elements is an AVL tree. In the case of *DocId* partitioning one bucket is used while 100 are currently used for *TermId*: words are hashed to a given bucket based on a dictionary (Cowie, 1989). The posting list is either created using the document identifier and the position record or updated by incrementing the number of positions and adding the position record to the position list. When any of the memory limits is reached, the results are saved on

a temporary file on disk for each bucket. A *worker* then requests work from the *farmer* and waits for a new document/file to analyse. A termination notice is received when there are no more documents/file to be processed and the *worker* either saves the inverted file directly from memory if the inversion has fitted into memory, or merges the intermediate results to create the inverted file. Where *DocId* partitioning is required the process can stop here, if *TermId* is required then a *global merge* is invoked.

```
Loop until termination notice received
        Receive a document/file from the farmer
        Analyse document/file -> index
        If memory limits exceeded at any point during analysis
                then save index on disk
        Send request for work to farmer
EndLoop
If memory limits have not been exceeded
        Save index directly to create inverted file
Else
        Save current index to disk.
        Merge data saved on disk to create inverted file
EndIf
```

Fig 4. Worker algorithm for parallel indexing

3.3 Global Merge Process

This further process is only used for *TermId* partitioning (see fig 5). The *global merge* process has three phases; a heuristic is applied to choose the distribution of the files, the files are then transferred across the network to the required node and a second *Local* merge is initiated to create the final inverted file. The heuristic in the first phase works by calculating the average value for each of the 100 partitions and attempts to derive a distribution of buckets amongst nodes that is within a given criterion, currently with 10% of the average value: up to five iterations are used. The average chosen for distribution is to prevent a node being overloaded with data, while iterations were restricted to ensure the process of allocating terms to nodes was fast. The average value can be one of three variables on a bucket; word count (WC), collection distribution (CF) and term distribution (TF): we refer to these as term allocation strategies. When the distribution is generated it is used to transfer the files for that bucket to the node that has been allocated that bucket: this is done by gathering from all processes to the target process. The merge is then initiated on those transferred files.

```
Worker i

(Phase 1)
Exchange word frequency data with all other workers
Partition words amongst workers using required word distribution
        criteria (WC,TF,CF)

(Phase 2)
Loop no of partitions -> j
        If partition j belongs to worker i
                gather partition j data from all other workers
        Else
                Send partition j data to required worker
        EndIf
EndLoop

(Phase 3)
Merge data for Workers partition to create inverted file.
```

Fig 5. Global merge algorithm for parallel indexing (*TermId* only)

4. *LOCAL BUILD* TOPOLOGY COMPONENTS

4.1 Timing Process

The only central process for *local build* is the timing process: it waits until all indexer processes are finished and saves the total elapsed time for the build. Our reasoning for using this method is to examine the scalability of our parallel data structures and algorithms: however because of its minimal communication it is the one most would choose in many circumstances.

4.2 Indexer Process

Each indexer process is a sequential index process that takes the function of the *farmer* and *worker* processes i.e. it reads in documents, breaks them down, adds them to the index creating intermediate indexes when a given set of criteria is met. The intermediate results are then merged to form one index for each node. The indexer process only communicates with the timing process when it has finished building the index: apart from that, its work is completely independent of any other process.

5. INDEXING METHODOLOGY

For each index build we used a stop word list of 450 words supplied by Fox (1990) to filter out unwanted terms. All HTML/SGML tags are stripped from the text and ignored if not used for specific reasons such as identifying paragraphs <p> and the end of document </DOC>. Each identified word was put through a Lovins stemmer, supplied by the University of Melbourne, and indexed in stem form. Numbers were not indexed. A large amount of in-core memory is pre-allocated in blocks by each indexing process, and documents are analysed until one of several criteria is reached: exhaustion of keyword block, posting block or position block space. When one of the criteria is satisfied, the current analysis is saved on disk as an intermediate index, so that the in-core memory can be used for the next set of documents. When all documents have been analysed, the intermediate indexes are merged together to create the final index and deleted.

6. HARDWARE USED

PLIERS (ParaLLel Information rEtrieval Research System) is designed to run on several parallel architectures and is currently implemented on those which use Sun Sparc, DEC Alpha and Pentium PII processors. All results presented in this paper were obtained on an 8 node Alpha farm and 8 nodes of a 12 node AP3000 at the Australian National

University (ANU), Canberra. Each node has its own local disk: that is a *shared nothing architecture* (DeWitt & Gray, 1992) is used by PLIERS. For the Alpha farm, each node is a series 600 266Mhz Digital Alpha workstation with 128 Mbytes of memory running the Digital UNIX 4.0b operating system. Two types of network interconnects were used: a 155 Mbytes/s ATM LAN with a Digital GIGASwitch and a 10 Mb/s Ethernet LAN: most of the indexing was done on ATM. The Fujistsu AP3000 is a distributed memory parallel computer using Ultra 1 processors running Solaris 2.5.1. Each node of the AP3000 has a speed of 167Mhz. The machine we used has 12 nodes, but only 8 are available on a partition. The torus network has a top bandwidth of 200 Mbytes/s per second.

7. DATA DESCRIPTION

We use a number of collections in our experiments: BASE1 and BASE10 plus BASE2, BASE4, BASE6 and BASE8 that are subsets of BASE10. BASE1 and BASE10 are officially defined samples of the 100 Gigabyte VLC2 collection (Hawking et al, 1999) and are 1 and 10 gigabytes in size respectively. The subsets of the official BASE10 collection were created by varying the number of BASE10 compressed text files put through the indexing mechanism (130 files per node for BASE2, 260 for BASE4, 390 for BASE6, 520 for BASE8). Each of the BASE **x** collections is approximately **x** gigabytes in size.

The strategy used to distribute the BASE1 and BASE10 collections for *local build* was to evenly spread the directories (in which the data is distributed by the ANU) among the nodes as far as possible. An alternative if more time consuming strategy is to do it by file size. The requirement of a distribution strategy is to get the best possible load balance for indexing as well as term weighting and passage retrieval search. The distribution process was done before the indexing program was started, and is not included in the timings.

Two types of inverted files were used for experiments: one type that recorded position information (necessary for passage retrieval and adjacency operations) and one that recorded postings only in the inverted list. The conventional form of inverted file was used with a clear keyword and postings file split. A document map was also used to store data such as document length: this file is fragmented with *local build* and replicated with *distributed build*. Map data on *distributed build* with *DocId* could be fragmented, but we chose to replicate rather than maintain extra source code in order to save time.

8. INDEX GENERATION TIME COSTS

In this section we declare the timing results on indexing using the configurations described above. The results are compared and contrasted where necessary as well as comparing them with available results for other systems on the BASE1 and BASE10 collections used in the VLC2 sub-track at TREC-7 (Hawking et al, 1999). We use the *local build* method on all defined collections, but only BASE1 is indexed using the *distributed build* method. The measures discussed are: indexing elapsed time in hours, throughput, scalability, scaleup, speedup and efficiency load imbalance (LI) and merging costs. Metrics used are defined in the glossary. Results on the Alpha farm and the AP3000 are discussed.

8.1 Indexing Elapsed Time

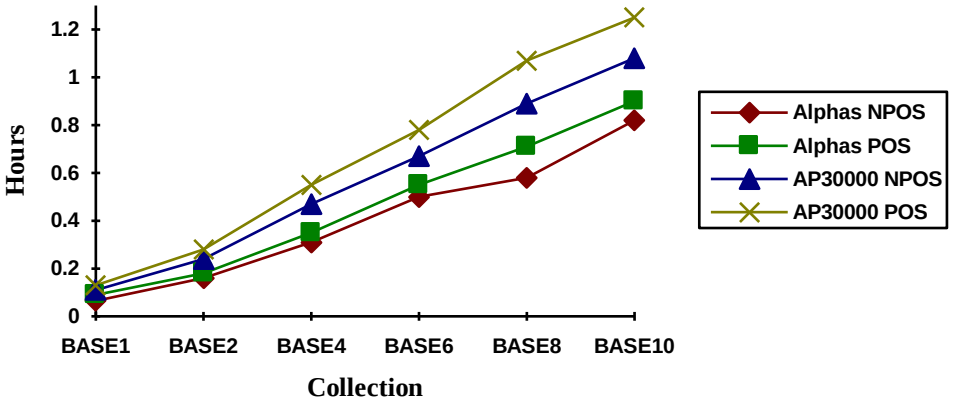


Fig 6. BASE1-10 *local build* [*DocId*]: indexing elapsed time in hours

In general, the Alpha farm was much faster than the AP3000 for indexing elapsed time as its processors are faster. For example on BASE10 *local build* indexing with postings only data took 0.82 hours on the Alphas and 1.08 hours on the AP3000 (see fig 6). The Alpha elapsed times recorded on *local build* also compare well with the results given at VLC2 (Hawking et al, 1999). That is, on BASE1 only two groups report slightly faster times than our posting only elapsed time of 0.065 hours (0.043 and 0.052 hours). Our sequential elapsed time on BASE1 at 0.56 (postings only) also compares well with those groups utilising a single processor: two other groups using uniprocessors recorded 0.42 and 1 hour respectively (refer to figs 7 and 8). On BASE10 on the Alphas the comparison is even more encouraging: only one group records a faster time of 0.504 hours. It should be noted that while the group with the fastest BASE10 indexing time uses a much smaller machine configuration (4 Intel PII processors) they use a very different method of inversion in which the collection is treated as one document (Clarke et al, 1998).

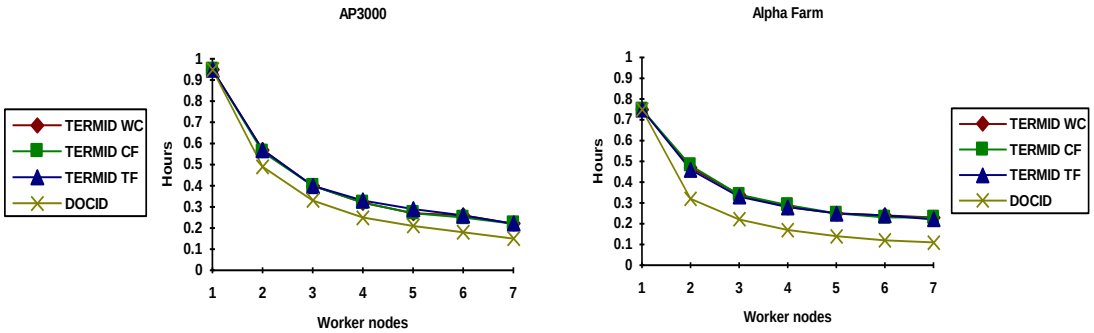


Fig 7. BASE1 *distributed build*: indexing elapsed times in hours (position data)*
(*Note: refer back to section 3.3 re: WC/CF/TF)

The results for *distributed build* indexing are presented in figs 7 and 8. The elapsed times for *DocId* are much better than those for the *TermId* method. This trend can be seen in all of the diagrams irrespective of machine or inverted file type used. The smallest difference is found on indexes with postings only using the AP3000. In general *TermId* elapsed times were longer than *DocId* because of the amount of data that has to be exchanged between nodes for the method, particularly for indexes with position data. Very little difference in time was found in any of the term allocation strategies (see section 3.3) studied for *TermId*.

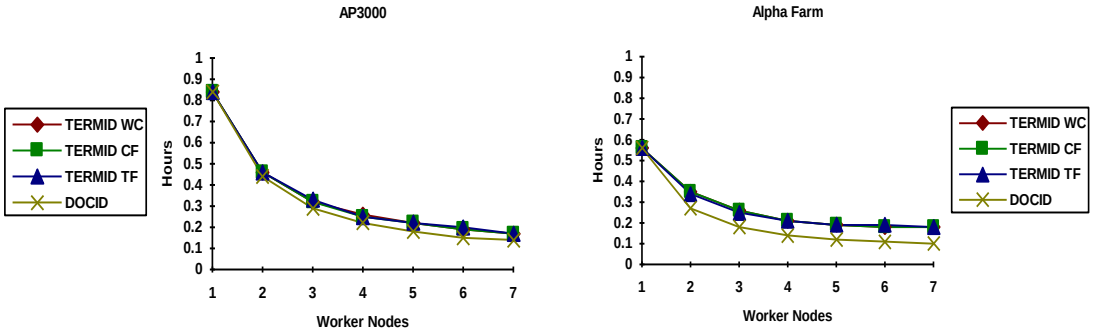


Fig 8. BASE1 *distributed build* : indexing elapsed times in hours (postings only)

One interesting factor found in the *TermId* results was that the AP3000 outperformed the Alpha farm at 7 worker nodes largely due to the extra network bandwidth available. It is that this point where the compute/communication balance favours the AP3000. A further run using *distributed build* with *DocId* partitioning on the Alpha farm revealed how much faster it is to use the ATM network than the Ethernet network: the time with ATM on 2 worker nodes building an index for BASE1 with no position data was 0.27 hours, while the figure for Ethernet was nearly double at 0.47 hours. This comparison further illustrates the importance of network bandwidth to the *distributed build* method and which can cause problems in many IR tasks (Rungsawang et al, 1999). We did not conduct any further experiments on this type of build for indexing using the Ethernet network as a consequence.

The extra time costs engendered by generating inversion with position data varied (this ratio is declared in the glossary - our aim is to record a ratio as close to 1.0 as possible). For example, in *local build DocId* the difference between posting only generation and position data generation ranged between 1.09 - 1.37 times on the Alphas (where merging was required). The extra costs on BASE1 are the highest (1.25 for the AP3000 and 1.37 for the Alphas) because the index with postings only is saved directly to disk without the need for merging: merging is required only when memory limits have been exceeded. Fig 9 shows the ratios for *distributed build* experiments. How much these extra costs are justified depends on the query processing requirement: such as a user need for passage retrieval or proximity operators.

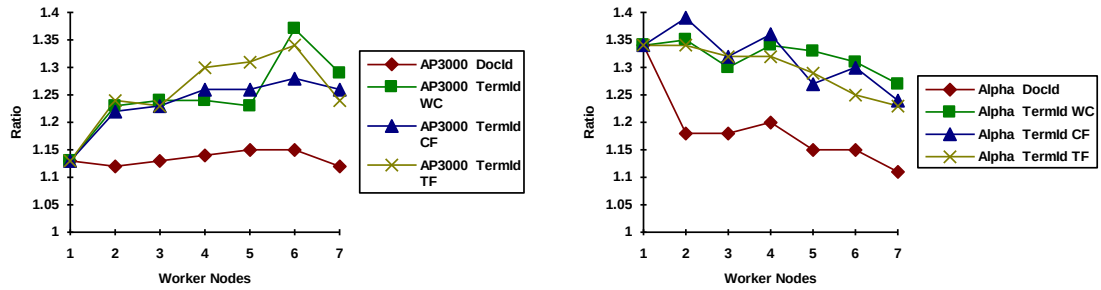


Fig 9. BASE1 *distributed build*: indexing extra costs for storage of position data

8.2 Throughput

The metric we use for throughput is Gigabytes of text processed per hour (G/Hour) to compare performance between database builds. Fig 10 shows the throughput for 8 processor configurations. The throughput for the Alphas is much faster than for the AP3000, e.g. on BASE1 *local build* indexing with postings only the rate is 15.4 G/Hour compared with 9.5 G/Hour on the AP3000. These are by far the best throughput results because no merging was needed: the configuration had enough memory to store the whole index and save it directly. The rate for other collections for *local build* indexing was 12-14 G/Hour on the Alphas for

postings only. Only one VLC2 participant recorded faster throughput for BASE1 and BASE10 collections (just over 19 G/Hour). The throughput on BASE1 using *distributed build DocId* with is not as good the *local build* but is still encouraging (see fig 11).

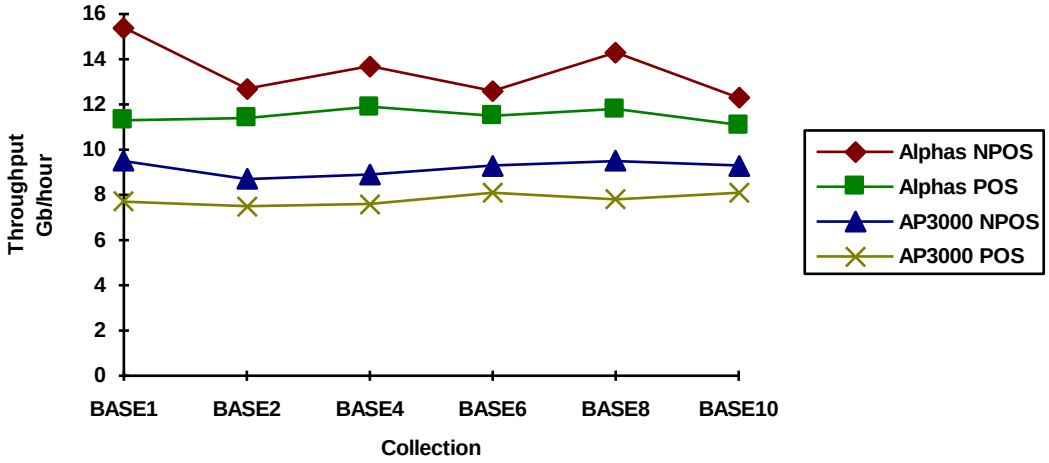


Fig 10. BASE1-BASE10 *local build* [*DocId*]: indexing Gb/Hour throughput

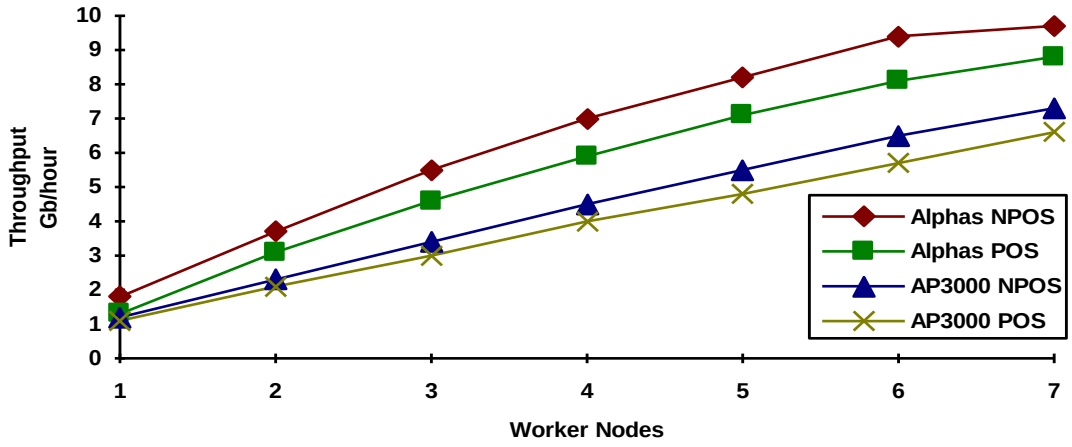


Fig 11. BASE1 *distributed build* [*DocId*]: indexing Gb/Hour throughput

It was found that increasing the number of worker nodes increased the throughput for both *distributed build* methods. For example, the *DocId* results for 7 worker nodes yielded a throughput of 9.7 G/Hour on the Alphas for postings only data indexes, compared with 1.8 for the uniprocessor experiment. The throughput for *TermId* builds was not as impressive but still acceptable with postings only: for example 5.8 G/Hour was recorded on the AP3000. The throughput for builds with position data was not as good, with 4.5 G/Hour on the AP3000 (see fig 12). Note that we only declare results for *TermId* with the word count (WC) method as there is very little difference in measurement between any of the term allocation strategies studied. Note also the superior performance in throughput on the AP3000 at 7 worker nodes due to the extra bandwidth available with that machine.

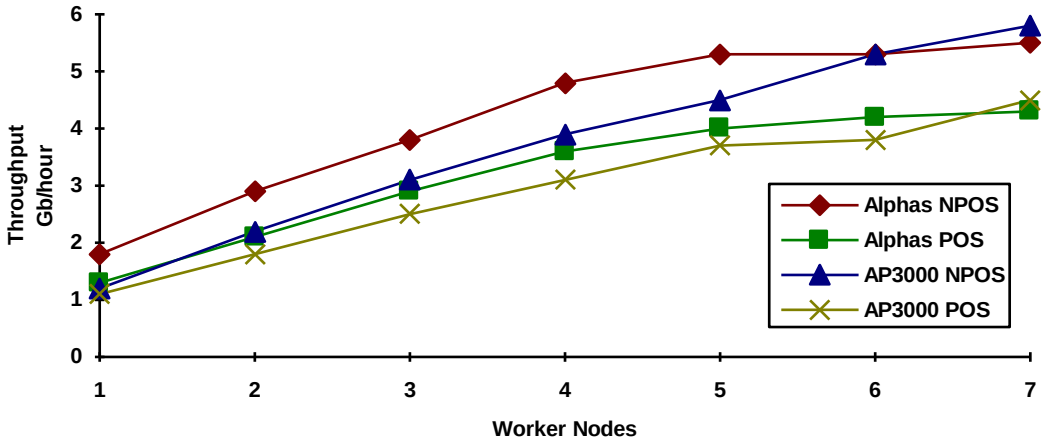


Fig 12. BASE1 *distributed build* [*TermId*]: indexing Gb/Hour throughput (WC only)

8.3 Scalability

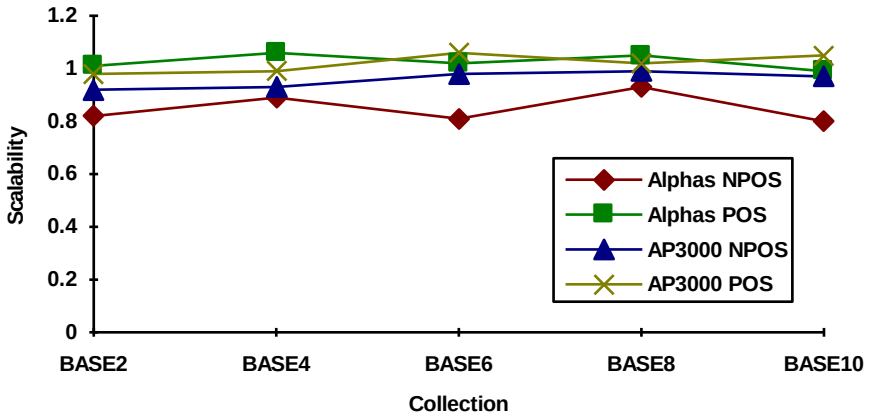


Fig 13. BASE2-BASE10 *local build* [DocId]: indexing scalability from BASE1

The data measure used in the equation is the size of indexed text. The scalability metric is defined in the glossary. We measure the effect of increasing collection size on the same sized parallel machine using the BASE2-10 collections over the BASE1 collection. We look for a scalability of around 1.0, greater than 1.0 being the aim. The results are presented in fig 13. With postings only data the scalability ranges between 0.80 and 0.93 on the Alphas and 0.92 and 0.99 on the AP3000. These figures are rather distorted because of the direct save on BASE1, that is no merging was needed as memory limits were not exceeded. The results are on the pessimistic side (if more memory was available we might be able to save indexes directly on all the collections studied). In builds with position data the scalability is excellent with the Alphas registering super-linear scalability on most BASEx (BASE10 was the exception) and the AP3000 delivering super-linear scalability on BASE6,8 and 10. The scalability results for indexes with position data demonstrate that the algorithms and data structures implemented are well able to cope with the extra computational load and data size that such builds both require and process.

8.4 Scaleup

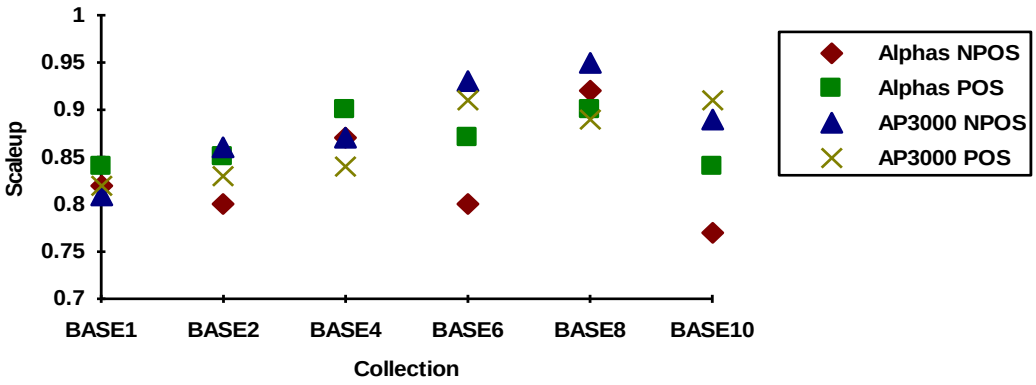


Fig 14. BASE1-BASE10 *local build* [*DocId*]: indexing scaleup

The scaleup metric is declared in the glossary. We measure within BASEx scaleup for *local build* only in this section. We take the times on each individual processor and compare the smallest elapsed time with the largest elapsed time on all 8 nodes. We are comparing the smallest sub-collection of BASEx (1/8th of BASEx) with the full sized BASEx collection. We use the least favourable figure in our measurement to obtain the lowest scaleup from any of the chosen sub-collections: our measurements are therefore pessimistic. We look for a scaleup of around 1.0, greater than 1.0 being the aim. The results are given in fig 14. In general the scaleups recorded are very good with most above the 0.8 mark. The worst scaleup was measured over the BASE10 collection on builds with no position data with a figure of 0.77. This figure was found on the Alpha farm where the processors are much faster. A combination of data size and processor speed can have an impact on scaleup: the scaleup figures for indexes with position data on the Alpha farm are generally superior to indexes without such data. The situation is reversed for AP3000 where the processors are slower. These scaleup figures show that there is little deterioration in performance of our implemented data structures and algorithms when moving from a smaller collection indexing on a small configuration parallel machine, compared with a larger collection on a larger configuration machine.

8.5 Speedup and Efficiency

All figures relate to the BASE1 collection. Definitions of these metrics can be found in the glossary. Recall that our ideal speedup is equal to the number of nodes, whereas for efficiency we look for a figure of 1.0. A surprising feature was the superlinear speedup and efficiency figures found with some of the indexing experiments particularly for the *local build DocId* 8 processor runs (see table 1). For example with the direct save on postings only data *local build* on the Alphas yielded a speedup of 8.5 and efficiency of 1.07. This effect was also found on some of the runs using *Distributed DocId* indexing (see figs 15 and 16).

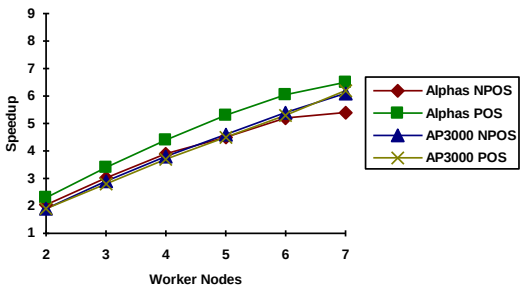


Fig 15. BASE1 *distributed build* [*DocId*]: indexing speedup

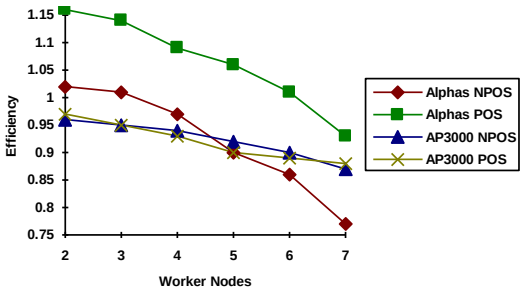


Fig 16. BASE1 *distributed build* [*DocId*]: indexing efficiency

Machine	File Type	Speedup	Efficiency
Alpha	NPOS	8.5	1.07
	POS	8.4	1.04
AP3000	NPOS	7.96	0.99
	POS	7.2	0.90

Table 1. BASE1 *local build* [*Docid*]: indexing speedup and efficiency

The reason this effect can occur is the extra memory multiple nodes have compared with a sequential processor, i.e. on *local build* with 8 nodes the index fits into main memory and it can be saved directly without the need for merging. More memory reduces the number of intermediate results saved to disk and therefore saves I/O time when data is merged to create the index. On *distributed build* a two *worker* configuration has twice the memory of the sequential program. The super-linear effect tails off at various stages on the *Distributed* version as communication time becomes more important (see fig 15).

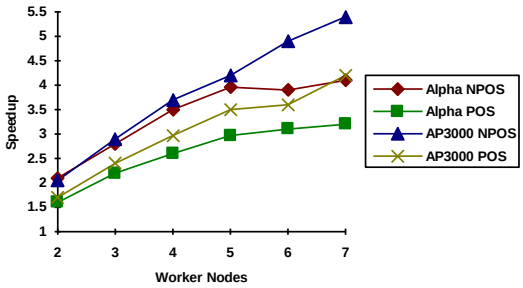


Fig 17. BASE1 *distributed build* [*TermId*]: indexing speedup (WC only)

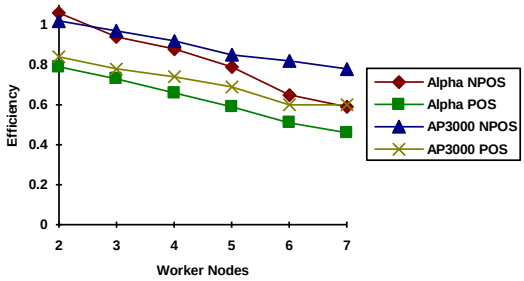


Fig 18. BASE1 *distributed build* [*TermId*]: indexing efficiency (WC only)

With *TermId* communication is very important: the global merge reduces most speedup/efficiency measures to less than linear (see figs 17 and 18). With position data and *TermId* there is little speedup on the Alpha Farm and efficiency ranges from the average to poor. Interestingly super-linear speedup/efficiency does occur on two worker nodes with builds on posting only data: further evidence of the significance of the memory effect.

8.6 Load Imbalance

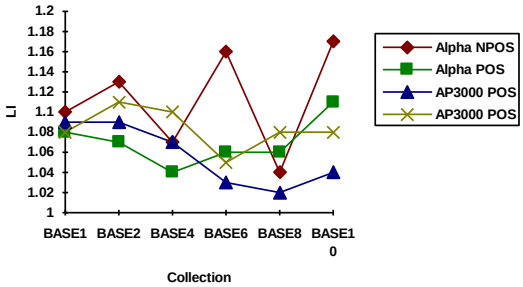


Fig 19. BASE1-BASE10 *local build* [*DocId*]: indexing load imbalance

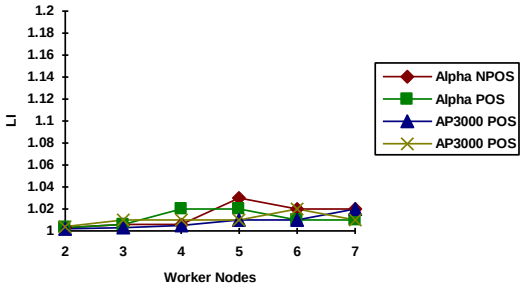


Fig 20. BASE1 *distributed build* [*DocId*]: indexing load imbalance

The load imbalance metric we use is declared in the glossary - the ideal load balance is close to 1.0. In general it was found that the *distributed build* imbalance was lower than those of *local build* (see figs 19 and 20). In fact *distributed build* using any partitioning method is excellent on all nodes with both methods, e.g. on 2-7 Alpha and AP3000 *worker*s the LI was in the range 1.002 to 1.03 on average for *DocId*. The LI figures demonstrate that the implemented process farm method provides good load balance for indexing jobs when whole files are distributed to workers.

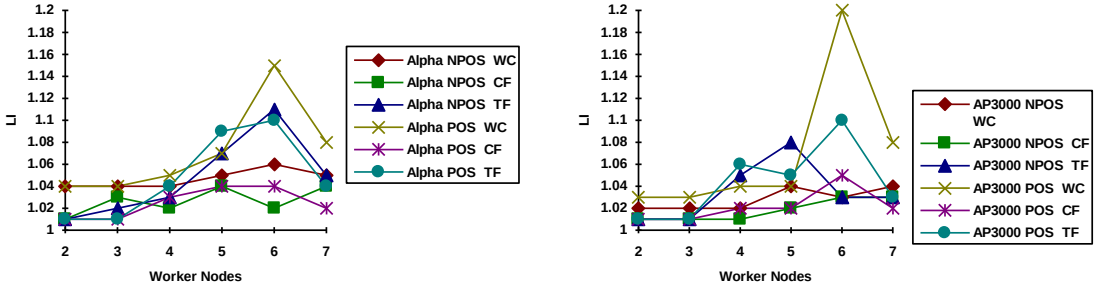


Fig 21. BASE1 *distributed build* [*TermId*]: indexing load imbalance

The results for *TermId* were generally not as good as *DocId*, but good in the average case (see fig 21). The exception was for builds with position data on 6 nodes: LI's of 1.2 for the AP3000 and 1.15 for the Alphas were recorded with word count (WC) distribution. The farm method described in, section 3 above is a very good way of ensuring load balance in the majority of cases. The *local build* LI is still very good: the worst LI recorded was 1.17 for BASE10 for the Alpha postings only run. We conclude by stating that both *Distributed* and *local build* methods achieve good load balance, but *local build* LI could be improved by paying more attention to text distribution.

8.7 Merging Costs

We consider here the percentage of time spent merging the temporary results to create the final inverted file: see the glossary for a formal definition - we look for the lowest possible cost in % terms. We examine the *DocId* method first. The merging for *local build* was in the main consistent within a 1% range, e.g. on the Alphas with posting data only, the average merge cost was 14 to 15% (see table 2). Merging costs for builds with position data were higher, e.g. on the Alphas the merge cost was 19 to 20%. Merge costs on the AP3000 were lower on *local build*, e.g. with posting data the average merge cost was around 13 to 14%. This difference is because the Alpha Farm processors are much faster and therefore the I/O time (which remains constant) is more significant.

With *distributed build DocId* build the merging costs were much the same as *local build* apart from Alpha builds with position data: the range found was 17 to 20%: these costs did not vary much from the *local build* (see table 3). The uniprocessor builds with position data registered the highest merge costs, whereas parallel *DocId* builds without position data saved indexes directly without the need for merging on 8 processors. The merge costs were more prominent on the Alpha as the faster processor speed reduces the computational costs

and increases the importance of I/O (merge is an I/O intensive process). Merge costs are also more prominent on indexes which contain position data.

Collection	Alphas		AP3000	
	NPOS	POS	NPOS	POS
BASE1	-	20%	-	14%
BASE2	14%	19%	10%	14%
BASE4	14%	19%	9%	13%
BASE6	15%	19%	9%	13%
BASE8	14%	19%	9%	13%
BASE10	14%	19%	9%	14%

Table 2. BASE1-10 *local build* [*DocId*]: % of average elapsed indexing time spent merging

Work-ers	Alpha		AP3000	
	NPOS	POS	NPOS	POS
1	15%	24%	9%	16%
2	15%	20%	9%	14%
3	15%	19%	10%	14%
4	15%	20%	10%	14%
5	15%	19%	10%	13%
6	14%	18%	9%	13%
7	13%	17%	9%	14%

Table 3. BASE1 *distributed build* [*DocId*]: % of average elapsed indexing time spent merging

Work-ers	Alphas NPOS			Alphas POS			AP3000 NPOS			AP3000 POS		
Value	*WC*	*CF*	*TF*	*WC*	*CF*	*TF*	*WC*	*CF*	*TF*	*WC*	*CF*	*TF*
2	38%	37%	38%	44%	43%	44%	26%	26%	26%	35%	36%	37%
3	36%	35%	36%	42%	42%	42%	26%	26%	26%	34%	34%	34%
4	35%	34%	35%	41%	40%	40%	26%	26%	26%	34%	34%	41%
5	32%	31%	31%	36%	37%	36%	24%	25%	25%	31%	32%	38%
6	28%	29%	27%	33%	34%	33%	24%	24%	24%	29%	30%	30%
7	26%	26%	25%	30%	31%	31%	23%	23%	23%	28%	30%	29%

Table 4. BASE1 *distributed build* [*TermId*]: % of average elapsed indexing time spent merging: *distributed build*

Merge costs for *TermId* are very much higher as one would expect given the extra work required for merge with that method to exchange data between nodes (see table 4). These higher merge costs are a contributory factor in the overall loss of performance for *TermId* partitioning index builds. However there is a distinct decrease in all cases of the significance of merging on the Alphas, e.g. merging on indexes with position data and word count (WC) word distribution decreased from 44% at 2 *worker*s to 30% on 7 *worker*s. This is largely because the costs in transferring index data before the second merge can proceed increases with the numbers of worker nodes deployed, e.g. on the Alpha indexes with position data the increase is from 2 minutes at two *worker*s to 4 minutes at seven *worker*s. On the AP3000 a slight decrease in merging costs is recorded in most cases, and the decrease is not as pronounced as the Alphas. The Alpha's extra processor speed brings benefit to extra merging found when building *TermId* indexes. The corresponding figure for transferring indexes with position data on the AP3000 ranges from 2.4 minutes with two *worker*s to 2.9 with seven *worker*s. The AP3000 is better able to cope with this extra cost in transferring data for the second merge as it has extra bandwidth available in its network.

8.8 Summary of Time Costs for Indexing

With respect to comparable metrics such as elapsed time and throughput, we have demonstrated that for a least one partitioning method, namely *DocId*, our results are state of the art compared with other VLC2 participants (Hawking et al, 1999). We have found that in most cases the Alpha farm outperforms the AP3000 except for some *TermId* runs: the AP3000 has a much higher bandwidth network available to it that is an advantage in such builds. Comparing the partitioning methods we have found that builds using the *DocId* method outperform index builds using *TermId* in all experiments. Our speedup and efficiency figures show that the methods of parallelism do bring time reduction benefits, particularly for

the *DocId* partitioning method. The scalability and scaleup figures show that our implemented data structures and algorithms are well able to cope with increasingly larger databases on a same sized or larger parallel machine. The load imbalance is generally quite small for all runs. The extra costs for generating indexes with position data vary, but are not an insubstantial part of the overall costs. Merge costs are also an important element of total time, depending on the build and partitioning method used.

9. INDEX FILE SPACE COSTS

In this section we declare the space overheads using the configurations described above. The results are compared and contrasted where necessary as well as comparing them with overheads on the BASE1 and BASE10 collections used in the VLC2 sub-track at TREC-7 (Hawking et al, 1999). The space overheads discussed are: overall inverted file space costs, keyword file space costs and file space imbalance.

9.1 Inverted File Space Costs

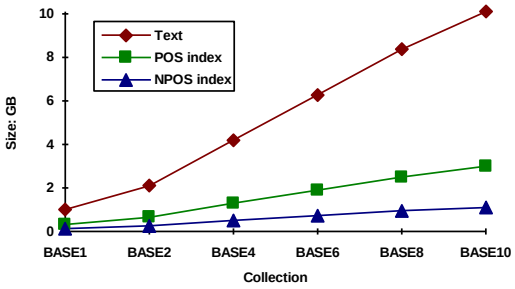


Fig 22a. BASE1-BASE10 *local build* [*DocId*]: index space costs in Gigabytes

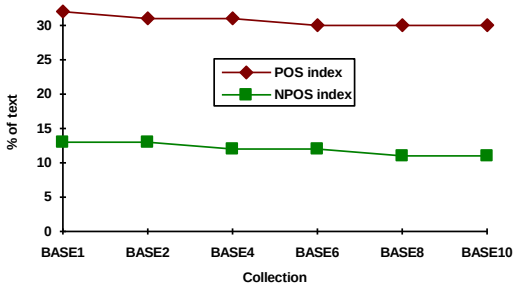


Fig 22b. BASE1-BASE10 *local build* [*DocId*]: index space costs in % of text

The metrics we used here are the file sizes in Gigabytes and percentage of original text size. The space costs for *local build* indexes are fairly constant in percentage terms across all collections (see fig 22b), although a slight reduction in index size compared with the size of the text can be see in fig 22a. This reduction occurs irrespective of the type of data stored in the inverted file. From fig 23 we can observe that there is a slight increase in index size for increasing the processor set when using *distributed build* methods. The reason for this is because of the replicated map requirements of *distributed builds*. The increase is more marked for *DocId* partitioning. If the map file size is taken away from the total size then the *DocId* indexes increase is much smaller (the reason any increase at all is explained in section 9.2).

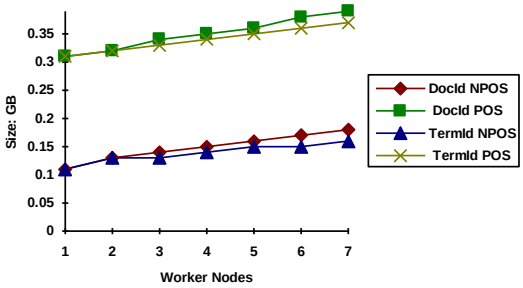


Fig 23a. BASE1 *distributed build*: index space costs in Gigabytes

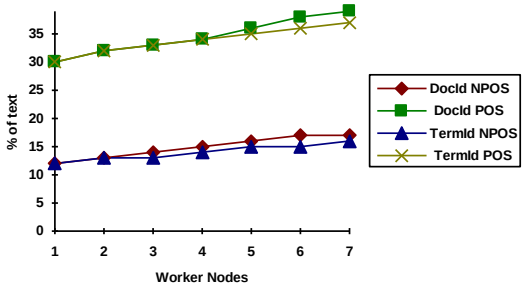


Fig 23b. BASE1 *distributed build*: index space costs in % of text

The comparison with space costs of the VLC2 participants (Hawking et al, 1999) is favourable with postings only data: our smallest figure of 0.11 Gigabytes on BASE1 was smaller than all submitted results and on BASE10 only one VLC2 participant at 0.902

Gigabytes was smaller than our figure of 1.1 Gigabytes. The comparison with files that contain position data is not so good and our smallest figure of 0.31 Gigabytes for BASE1 is bested by two groups, while on BASE10 three groups record a smaller figure than our 3.0 Gigabytes.

9.2 Keyword File Space Costs

The metric for keyword file space costs is the size in megabytes and the keyword file percentage of the total inversion. With *local build* on both postings only and position data we found that the trend in keyword space costs was a decreasing one, e.g. 32% on BASE1 to 22% on BASE10 with postings only data (see fig 24b). This is because the increase in lexicon is not linear with the increase in collection (fig 24a). With *distributed DocId* indexes the keyword costs remain constant, e.g. 24-26% (see fig 25b). The size of the keyword file actually increases with more inverted file partitions (see fig 25a), but this increase is not significant and is absorbed by the increase in size of the replaced document map. We state that there is little extra cost in having words replicated across different fragments for *DocId* partitioning on this type of collection (Web data). For *TermId* indexes the size of the keyword file was constant irrespective of term allocation method, and if the map data is included in costs the significance of the keyword file with respect to the total index size gradually decreases (see figs 25a and 25b).

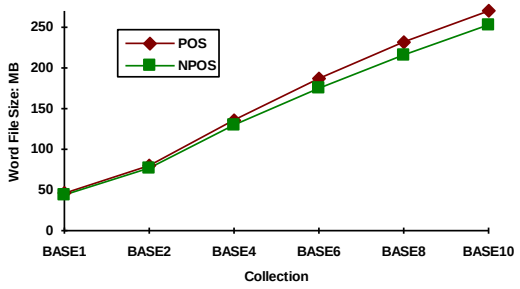


Fig 24a. BASE1-BASE10 *local build* [*DocId*]: index space costs in megabytes for keyword file

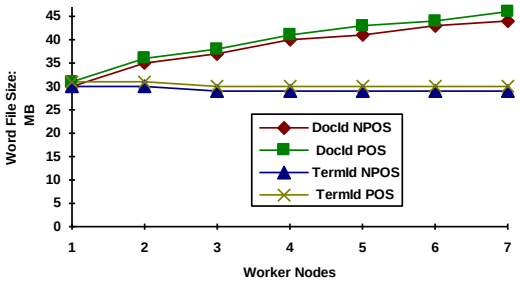


Fig 25a. BASE1 *Distributed Build*: space costs in megabytes for keyword file

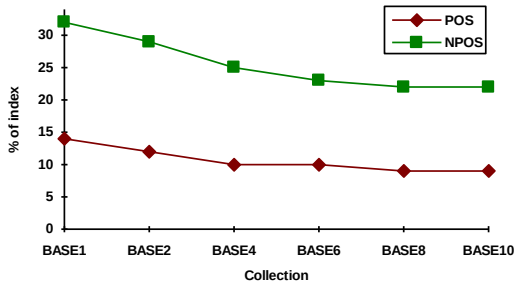


Fig 24b. BASE1-BASE10 *local build* [*DocId*]: index space costs in % of index for keyword file

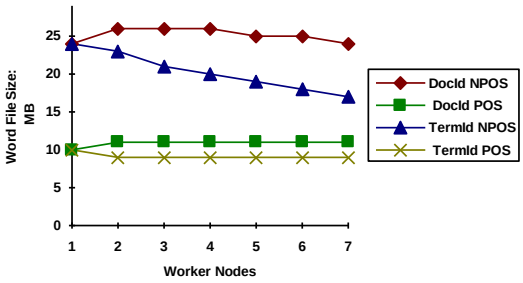


Fig 25b. BASE1 *Distributed Build*: index space costs in % of index for keyword file

9.3 File Load Imbalance

We use the concept of load imbalance (LI) but apply it to file sizes instead, i.e. maximum file size / average file size. We wish to ensure that index data is fairly distributed amongst nodes, e.g. it would not be desirable for one index partition to exceed the space available on a physical disk. The index time LI results are included in the figs 26 to 28 for comparative purposes. The space imbalance for text space costs was in general fairly stable being in the range 1.04 to 1.02 for all *local build* indexing runs (see fig 26). In comparison the inverted file imbalance was much higher, particularly for the smaller collections. Clearly the imbalance stems not from the size of the text, but from aspects of the text such as the

number of documents and total word length of the text. In contrast the space imbalance for *distributed build* on *DocId* partitioning was small for any type of inverted file data storage (see fig 27). There is no significant difference between the space imbalance of inverted files and LI for indexing times with *DocId* partitioning. The file space imbalance figures further proof of the validity of the farming method for balancing load for *DocId* partitioning.

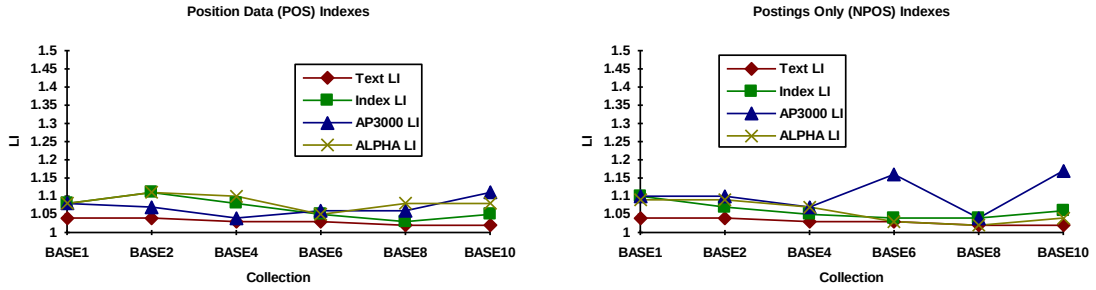


Fig 26. BASE1-BASE10 *local build* [*DocId*]: index space imbalance on files

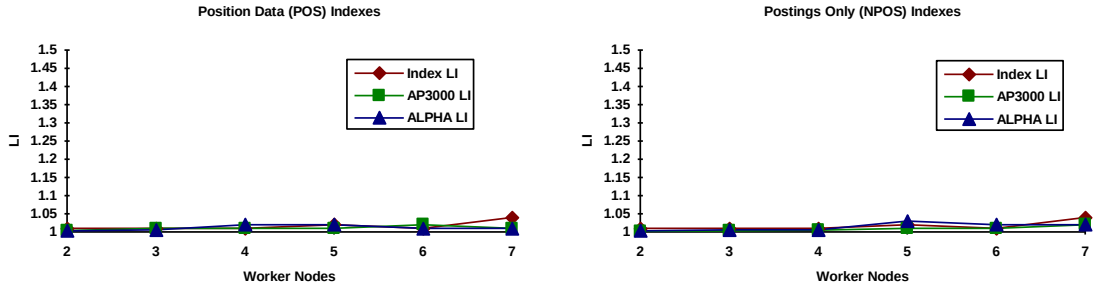


Fig 27. BASE1 *distributed build* [*DocId*]: index space imbalance on index files

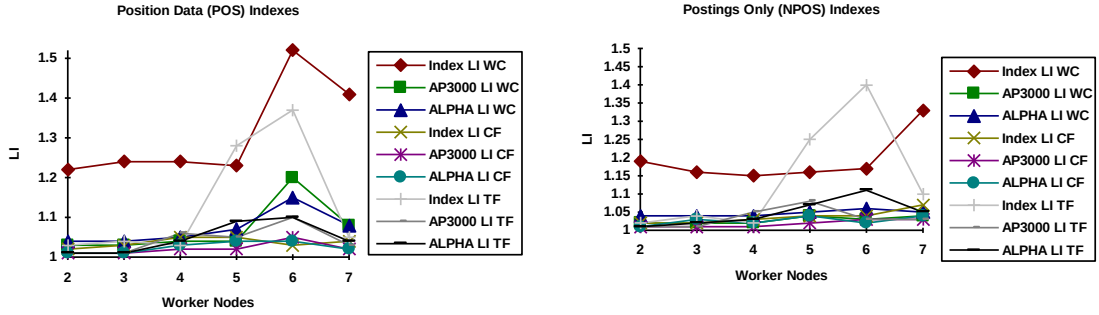


Fig 28. BASE1 *distributed build* [*TermId*]: index space imbalance on index files

The situation for *TermId* varies depending on the type of word distribution method used (see fig 28). For the word count (WC) distribution space imbalance was generally very poor, with the worst being indexes with position data on 6 worker nodes: an imbalance of 1.52 was recorded (interestingly the worst imbalance for indexing times, see fig 28). The figures for the collection frequency distribution method (CF) are much better with an imbalance range of 1.02 to 1.07 for all builds. In the term frequency (TF) method the imbalance was erratic being very poor at 5 and 6 worker nodes for any index builds, but good on all other runs. Any imbalance in space does not affect computational imbalance adversely. None of the *TermId* space imbalance results are as good as the *DocId* for space costs on *distributed builds*, as it is much harder to derive a good data distribution method for *TermId* indexes (the allocation of terms to nodes is a more difficult problem than allocating documents to nodes). None of the methods implemented affect space imbalance such that an index partition exceeds the physical disk of any node.

9.4 Summary of Space Costs for Indexing

Overall space overhead for the indexing is state of the art and comparable with the results give by VLC2 participants: at least for indexes with postings only. The *Distributed Build DocId* results show that the cost of storing keywords does grow with increasing the fragmentation, but given that *local build* results show that space costs decrease with database size we do not see this a serious overhead for the *DocId* partitioning method. The space costs imbalance for *local build* is generally quite stable, but the generated inverted files vary more. Clearly the consideration of the number of files on its own is not sufficient to ensure very good balance. For *distributed builds* space imbalance was much smaller, except for some *TermId* indexes where distribution methods are more difficult to derive: no index partition exceeds the size of a node's local disk.

10. CONCLUSION

The results produced in this paper show that of the partitioning methods, *DocId* partitioning using any build has by far the most promise and would in most circumstances be the method chosen for indexing. This would be the case particularly if the collection under consideration needed frequent re-builds. We have used the *DocId* method to good effect in the Web track for TREC-8 on the full 100 Gigabyte VLC2 collection (MacFarlane et al, 2000). Where disk space was limited, the *local build* method could be used to good effect: we used this build method on the BASE10 as we did not have sufficient space to do *distributed builds* on that collection. We have demonstrated that indexing is state of the art in both compute and space terms by comparing our space and time results with those given at VLC2 (Hawking et al, 1999) and the TREC-8 Web Track (Hawking et al, 2000). Although we did not produce the best results for all measures, no group at VLC2 did either. Our indexing time for the full 100Gb collection was the best in the Web Track (MacFarlane et al, 2000).

A clear distinction must be made between *DocId* and *TermId* partitioning methods. *Distributed build DocId* out-performs *TermId* in all areas of time cost metrics and would therefore always be preferred if indexing was of primary concern. We state this irrespective of the type of inversion or algorithms/methods used if cluster computing is utilised. We would recommend that *TermId* only be used if two main criteria are met. One is that a high performance network is available to reduce time spent on transferring data during the *global merge* process. The other is that some other benefit must accrue from the use of *TermId* partitioning which in essence would be some advantage in search performance or index maintenance criterion over the *DocId* method.

11. Acknowledgements

This work is supported by theArts and Humanities Research Board (AHRB) under grant number IS96/4203. We are also grateful to ACSys for awarding the first author a visiting fellowship at the Australian National University in order to complete this research and use of their equipment. We are particularly grateful to David Hawking for making the arrangements for the visit to the ANU.

REFERENCES

Bowler, K. C., Kenway, R. D., Pawley, G. S., Roweth, D & Wilson, G. V. (1989). An introduction to Occam-2 programming: 2nd Edition, Chartwell-Bratt.

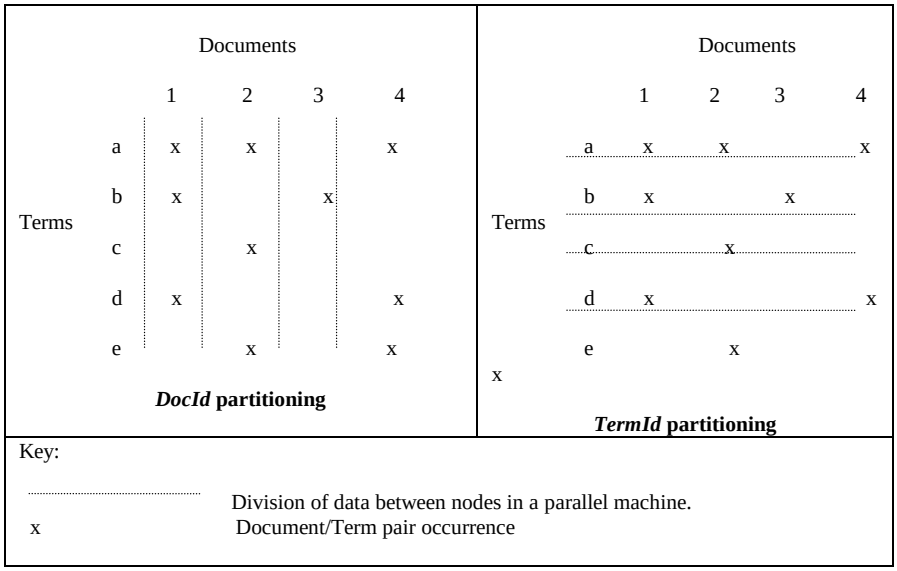
C.L.A.Clarke, G.V.Cormack & C.R.Palmer, An overview of MultiText, SIGIR Forum Vol 32, No 2, Fall 1998, 14-15.

Cowie, A.P. (Ed), (1989). Oxford Advanced Learner's Dictionary of current English, Fourth Edition, Oxford University Press.

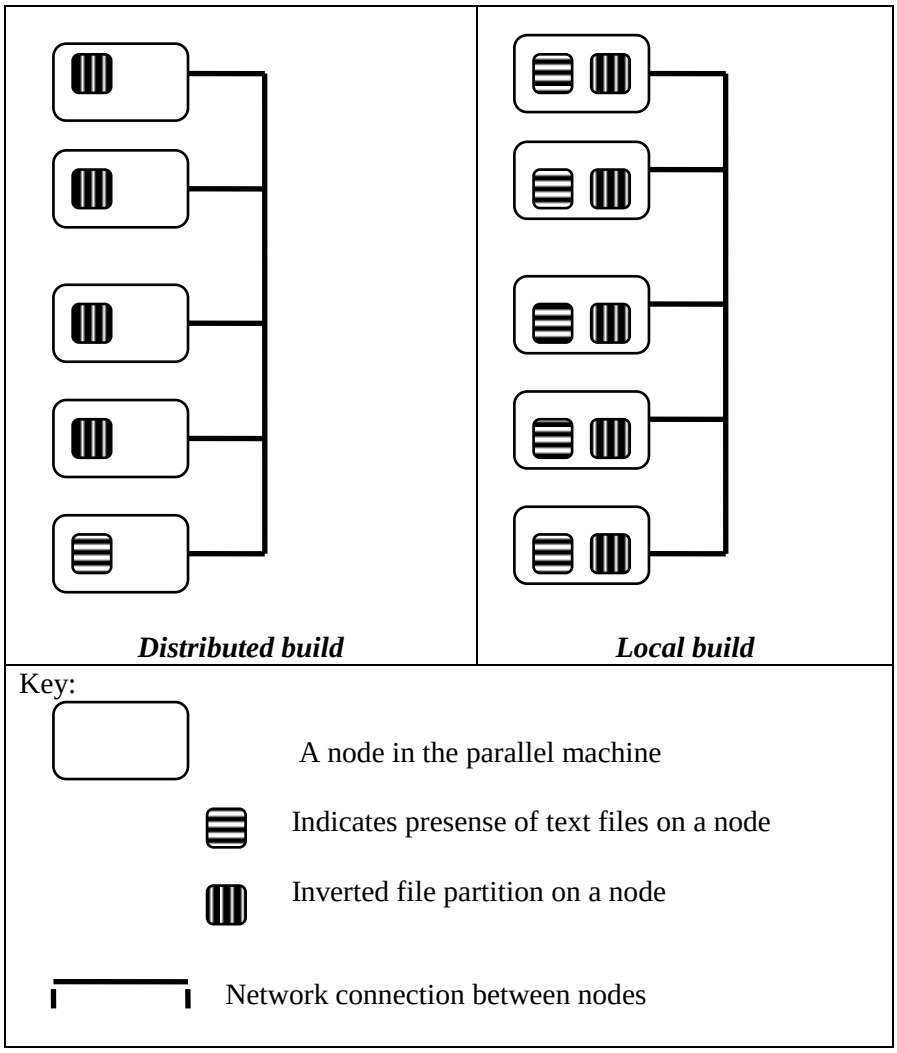
DeWitt, D. & Gray, J. (1992). Parallel database systems: the future of high performance database systems, Communications of the ACM, Vol 35, No 6, 85-98.

Fox, C. (1990). A stop list for general text, SIGIR FORUM, ACM Press, Vol 24, No 4,19-35.

Hawking D. (1995). *The design and implementation of a parallel document retrieval engine*. Technical Report TR-CS-95-08, Department of Computer Science. Canberra: Australian National University.
Hawking, D. (1997). Scalable text retrieval for large digital libraries. In: C. Peters and C. Thanos, eds., *Proc. first European Conference on Digitial Libraries*, Vol 1324, LNCS, Spinger-Verlag, 127-146.
Hawking, D., Craswell, N., & Thistlewaite, P. (1999). Overview of TREC-7 Very Large Collection Track, In: D.K.Harman, ed, *Proceedings of the Seventh Text Retrieval Conference, Gaithersburg, U.S.A, November 1998*, Gaithersburg, NIST SP 500-242, 257-268.
Jeong, B., & Omiecinski, E. (1995). Inverted file partitioning schemes in multiple disk systems, *IEEE Transactions on Parallel and Distributed Systems*, *6* (2), 1995, 142-153.
MacFarlane, A. (2000). Distributed Inverted files and performance: a study of data distribution methods and parallelism in IR, PhD Thesis, City University.
MacFarlane, A., Robertson, S.E., & McCann, J.A. (1997). Parallel computing in information retrieval - an updated review, Journal of Documentation, Vol. 53, No. 3, 274-315.
MacFarlane, A., Robertson, S.E., & McCann, J.A. (1999). PLIERS at VLC2, In: D.K.Harman, ed, *Proceedings of the Seventh Text Retrieval Conference, Gaithersburg, U.S.A, November 1998*, Gaithersburg, NIST SP 500-242, 327-336.
MacFarlane, A., Robertson, S.E., & McCann, J.A. (2000). PLIERS AT TREC8, In: E. Voorhess, ed, *Proceedings of the Eight Text Retrieval Conference, Gaithersburg, U.S.A, November 1999*, Gaithersburg, NIST SP 500-246, 241-252.
Rungsawang, A., Tangpong, A. & Laohawee, P. (1999). Parallel DISR text retrieval system. In: Dongarra, J., Luqueu E. and Margalef, T., eds. Proceedings of 6th European PVM/MPI Users' Group Meeting, Barcelona, Lecture Notes in Computer Science 1697, (Berlin: Springer-Verlag): 325-332.



Appendix 1 - An example of how Partitioning methods for Inverted Files distributes data



Appendix 2. Examples of build methods for distributed inverted files

Glossary

CF allocation	Method of term allocation in *TermId* to partition using a collection frequency criterion.
Distributed Build	Method of building indexes where text is distributed from a single node.
DocId	Parititioning method which assigns all document data for a given document to one index partition
Efficiency	Measure of the effective use of processors. Definition: Speedup on n processors/n processors
Elapsed time	Time to build an index.
Farmer	Process which distributes text to nodes.
Global Merge	Process which exchanges data between nodes in order to create a distributed *TermId* inverted file.
Indexer process	Process in local build which analyses text and builds inverted file to the local disk.
LI	A measure of the amount of load imbalance on n processors: max time on n processors/average time on n processors
Local Build	Method of indexing where all processing is kept local to the node.
Merge Costs	Percentage of time spent merging over all the processors. Definition $\frac{\text{average merging time on all P Processors}}{\text{average elapsed time on all P processors.}} * 100$
Mhz	Megahertz: processor clock speed.
Partition	Fragment of Inverted file on a nodes disk.
Position Data Extra Cost	$\text{Ratio} = \frac{\text{Elapsed Time for a given task on an index with position data}}{\text{Elapsed Time for the same task on an index with postings only}}$
Scalability	A measure of how well the algorithm scales on the same equipment. Definition: $\frac{\text{Time on small collection}}{\text{Time on large collection}} * \frac{\text{Size of large collection}}{\text{Size of small collection}}$
Scaleup	We define scaleup as the comparison metric [11]: $\frac{\text{elapsed time on P processors indexing small problem DB}}{\text{elapsed time on P' processors indexing big problem DB'}}$ where P' > P and DB' > DB
Speedup	Measure of speed advantage of parallelism. Definition: Time on 1 processors / Time on n processors.
TermId	Parititioning method which assigns all term data for a given term to one partition
TF allocation	Method of term allocation in *TermId* to partition using a term frequency criterion.
Timing Process	Process which times local build indexing elapsed time.
Throughput	Gigabytes of text processed per hour.
TREC	Annual Text Retrieval Conference run by the National Institute of Standards and Technology in the United States.
VLC	Very Large Collection: Collection of 100 GB web data used in the TREC-7 VLC2 sub-track.
WC allocation	Method of term allocation in *TermId* to partition using a word count criterion.
Web Track	Sub track of TREC-8.
Worker	Process which creates index data from raw text.
Zipf distribution	Distribution which suggests that a few words will occur in many documents, while many words will occur in few documents.